Language with Variable Local Binding

Substitution Version · AST · Parser · Substitution · Interpreter · Runner

---

1. Grammar (BNF)

The surface language supports numeric literals, addition, a single-variable local binding form, and variable references.

```
<expr> ::= <num>
       | {+ <expr> <expr>}
       | {let1 {<var> <expr>} <expr>}
       | <var>
```

2. AST

The **Exp** type represents all possible expressions. Constructors are suffixed with **E**.

```
(define-type Exp
  [numE  (n    : Number)]
  [plusE (left : Exp) (right : Exp)]
  [varE  (name : Symbol)]
  [let1E (var      : Symbol)
         (val      : Exp)
         (body     : Exp)])
```

3. Parser (S-Exp -> Exp)

Converts surface syntax into an **Exp** AST node. Identical to the environment version — the parser only builds the AST.

```
(define (parse [s : S-Exp]) : Exp
  (cond
    [(s-exp-number? s)
     (numE (s-exp->number s))]

    [(s-exp-symbol? s)
     (varE (s-exp->symbol s))]

    [(s-exp-list? s)
     (let ([l (s-exp->list s)])
       (cond
         [(symbol=? '+ (s-exp->symbol (first l)))
          (plusE (parse (second l))
                 (parse (third  l)))]

         [(symbol=? 'let1 (s-exp->symbol (first l)))
          (let* ([binding  (s-exp->list (second l))]
                 [var-name (s-exp->symbol (first  binding))]
                 [var-val  (parse         (second binding))]
                 [body     (parse         (third  l))])
            (let1E var-name var-val body))]

         [else (error 'parse "unknown operator")]))]

    [else (error 'parse "not a number, symbol, or list")]))
```

4. Substitution (Exp Symbol Exp -> Exp)

subst rewrites the AST by replacing every free occurrence of **name** inside **expr** with **val**. This is the key mechanism that eliminates variables — no environment table is needed.

Key rule for **let1E**: if the let binds the *same* name, substitute in **val-expr** but stop at **body** (the body lives in a new scope). If the name differs, substitute freely in both.

```
(define (subst [val  : Exp]
               [name : Symbol]
               [expr : Exp]) : Exp
  (type-case Exp expr

    [(numE n)
     (numE n)]

    [(plusE l r)
     (plusE (subst val name l)
            (subst val name r))]

    [(varE s)
     (if (symbol=? s name)
         val          ; replace this occurrence
         (varE s))]   ; different name -- leave alone

    [(let1E var val-expr body)
     (if (symbol=? var name)
         ; same name: substitute in val-expr, stop at body
         (let1E var (subst val name val-expr) body)
         ; different name: substitute freely in both
         (let1E var
                (subst val name val-expr)
                (subst val name body)))]))
```

5. Interpreter (Exp -> Number)

Pattern-matches on the AST and returns a raw **Number**. No environment is passed. Variables are eliminated by **subst** before any numeric computation. If a **varE** is reached, substitution missed it — the variable is unbound.

```
(define (interp [expr : Exp]) : Number
  (type-case Exp expr

    [(numE n)
     n]

    [(plusE l r)
     (+ (interp l) (interp r))]

    [(varE s)
     ; If we reach here, substitution missed it -> unbound.
     (error s "unbound variable")]

    [(let1E var val-expr body)
     ; 1. Evaluate val-expr to a number.
     ; 2. Wrap it as (numE ...) so subst can place it in body.
     ; 3. Substitute and interpret the result.
     (interp (subst (numE (interp val-expr))
                    var
                    body))]))
```

6. Runner (S-Exp -> Number)

Composes **parse** and **interp**. No environment is passed — the interpreter needs none.

```
(define (run [s : S-Exp]) : Number
  (interp (parse s)))
```

7. Tests

```
(print-only-errors #true)

; --- num and plus ---
(test (run `1)               1)
(test (run `{+ 1 2})         3)
(test (run `{+ {+ 1 2} 3})   6)

; --- simple let1 ---
(test (run `{let1 {x 1} x})         1)
(test (run `{let1 {x 5} {+ x x}})   10)
(test (run `{let1 {x 5} {+ x 1}})   6)

; --- nested let1, different names ---
(test (run `{let1 {x 1}
               {let1 {y 2}
                  {+ x y}}})
      3)

; --- shadowing: inner x shadows outer x ---
(test (run `{let1 {x 1}
               {let1 {x 2}
                  x}})
      2)

; --- shadowing: outer x still visible ---
(test (run `{let1 {x 1}
               {+ x
                  {let1 {x 2} x}}})
      3)

; --- unbound variable raises error ---
(test/exn (run `x)       "unbound")
(test/exn (run `{+ x 1}) "unbound")
```

Substitution vs. Environment — Key Differences

Variable resolution	`Rewrite AST with subst`	`Look up table with lookup`
varE in interpreter	`(error s "unbound variable")`	`(lookup s env)`
let1E in interpreter	`Calls subst, then interp`	`Calls extend, passes new env`
subst function	`Required`	`Gone entirely`
interp signature	`Exp -> Number`	`Exp Env -> Value`
Return type	`Raw Number`	`Value (numV)`

Environment	`None`	`mt-env, extend, lookup`
Shadowing mechanism	`subst stops at same-name let1`	`hash-set overwrites the key`

Pipeline Traces

Step-by-step execution of three example S-expressions through the full run -> parse -> interp pipeline using the substitution-based interpreter.

Trace 1 — `1

Step 1 | run is called

```
(run `1)
=> (interp (parse `1))
```

↓

Step 2 | parse receives `1

```
Input: 1
(s-exp-number? s)  =>  YES
(numE (s-exp->number s))
parse output:  (numE 1)
```

↓

Step 3 | interp receives (numE 1)

```
(interp (numE 1))
type-case matches:  [(numE n)  n]   where n = 1
A literal number needs no substitution whatsoever.
interp returns the raw number directly.
interp output:  1
```

↓

Final Output

```
1
```

Trace 2 — `{+ x 5}

Step 1 | run is called

```
(run `{+ x 5})
=> (interp (parse `{+ x 5}))
```

↓

Step 2 | parse receives `{+ x 5}

```
Input is a list:  l = [ '+,  x,  5 ]
                        first second third
head is '+  =>  plusE branch fires:
  parse (second l) = parse `x
    (s-exp-symbol? s)  =>  YES
    => (varE 'x)
  parse (third l) = parse `5
    (s-exp-number? s)  =>  YES
    => (numE 5)
parse output:  (plusE (varE 'x) (numE 5))
```

↓

Step 3 | interp receives (plusE (varE 'x) (numE 5))

```
plusE branch fires:
  (+ (interp (varE 'x))  (interp (numE 5)))
  -- left side first (Racket evaluates left-to-right) --
  (interp (varE 'x))
    type-case matches:  [(varE s)  (error s "unbound variable")]
    s = 'x
    subst never touched this varE  =>  x is unbound
    (error 'x "unbound variable")   *** STOP ***
```

↓

Error Raised

```
x: unbound variable The right side (numE 5) is never reached.
Racket evaluates left-to-right and stops at the first error.
No substitution was performed because there was no enclosing let1.
```

Trace 3 — `{let1 {x 1} {+ x 2}}

Step 1 | run is called

```
(run `{let1 {x 1} {+ x 2}})
=> (interp (parse `{let1 {x 1} {+ x 2}}))
```

Step 2 | parse receives `{let1 {x 1} {+ x 2}}

```
Input is a list:  l = [ let1,  {x 1},  {+ x 2} ]
                        first  second   third
head is 'let1  =>  let1 branch fires:
  binding  = (s-exp->list {x 1})  =>  [ x, 1 ]
  var-name = 'x
  var-val  = (parse `1)  =>  (numE 1)
  body     = (parse `{+ x 2})
           => plusE branch fires:
              (plusE (parse `x)   (parse `2))
              (plusE (varE 'x)    (numE 2))
parse output:
  (let1E 'x
         (numE 1)
         (plusE (varE 'x) (numE 2)))
```

Step 3 | interp receives let1E AST

```
(interp (let1E 'x (numE 1) (plusE (varE 'x) (numE 2))))
let1E branch fires:
  (interp (subst (numE (interp val-expr)) var body))
Three steps happen inside-out:
```

Step 3a | evaluate val-expr

```
(interp (numE 1))
  numE branch:  n  =>  1
result:  1
```

Step 3b | wrap result as numE

```
The number 1 must be wrapped back into an Exp
so that subst can place it inside the body AST.
(numE (interp val-expr))  =  (numE 1)
Wrapped value:  (numE 1)
```

Step 3c | subst — replace 'x with (numE 1) in body

```
(subst (numE 1) 'x (plusE (varE 'x) (numE 2)))
subst traverses the body AST:
  plusE branch:
    subst (numE 1) 'x (varE 'x)
      varE branch: symbol=? 'x 'x  =>  YES  =>  return (numE 1)
    subst (numE 1) 'x (numE 2)
      numE branch: no variable  =>  return (numE 2)
  result: (plusE (numE 1) (numE 2))
subst output:  (plusE (numE 1) (numE 2))
Note: the varE 'x is GONE. The AST has been rewritten.
```

↓

Step 3d | interp the rewritten AST

```
(interp (plusE (numE 1) (numE 2)))
plusE branch fires:
  (+ (interp (numE 1))  (interp (numE 2)))
  (+ 1                  2)
  3
interp output:  3
```

↓

Final Output

```
3
```

Summary of Pipeline Traces

`1	(numE 1)	n/a (no let1)	1
`{+ x 5}	(plusE (varE 'x) (numE 5))	n/a (no let1)	ERROR: x unbound
`{let1 {x 1} {+ x 2}}	(let1E 'x (numE 1) (plusE (varE 'x)(numE 2)))	(plusE (numE 1) (numE 2))	3